Programmeerimise algkursus

binaarne otsing, sorteerimine

Loengu sisu

- **Binaarne otsing**
- **Sorteerimine**
  - Aeglased sorteerimisalgoritmid: keerukus N*N
  - Merge sort: keerukus N*log N
  - Quicksort: hea keskmine keerukus, halb halvim juht
  - Võrdlusi ja näiteid

Binaarne otsing

- Tahvlil

Sorting

- Naive, simple sorting algorithms take

  N*N tries (quadratic time) in worst case

  Examples: bubble sort, insertion sort, selection sort

- Optimal algorithms (several meanings of “optimal”) take

  N*log(N) tries in worst case

  Examples: merge sort, heap sort

- Possibly infinite number of different sorting algorithms can be devised

- For example, “Quicksort” is bad on worst case, but very good on average case.

Insertion sort (slow): partial algorithm at first

```
static void insert(int[] A, int itemsInArray, int newItem) {

  int loc = itemsInArray - 1;  // Start at the end of the array.

  /* Move items bigger than newItem up one space;
     Stop when a smaller item is encountered or when the
     beginning of the array (loc == 0) is reached. */

  while (loc >= 0 && A[loc] > newItem) {
     A[loc + 1] = A[loc];  // Bump item from A[loc] up to loc+1.
     loc = loc - 1;        // Go on to next location.
  }
  A[loc + 1] = newItem;  // Put newItem in last vacated space
}
```

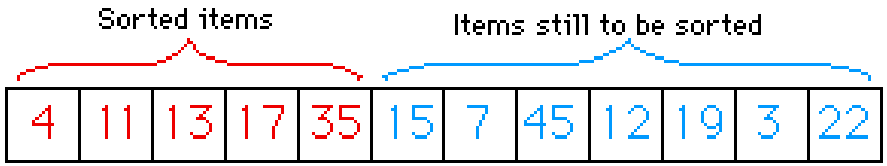
Insertion sort: full algorithm

- **Idea**: shift each next element to a right place in a sorted part of array.

```
static void insertionSort(int[] A) {
  int itemsSorted; // Number of items that have been sorted so far.
  for (itemsSorted = 1; itemsSorted < A.length; itemsSorted++) {
    int temp = A[itemsSorted];  // The item to be inserted.
    int loc = itemsSorted - 1;  // Start at end of list.
    while (loc >= 0 && A[loc] > temp) {
        A[loc + 1] = A[loc]; // Bump item from A[loc] up to loc+1.
        loc = loc - 1;       // Go on to next location.
    }
    A[loc + 1] = temp; // Put temp in last vacated space.
  }
}
```

Insertion sort

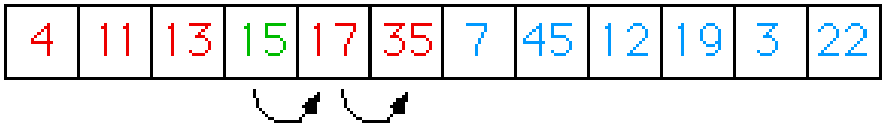
Start with a partially sorted list of items:

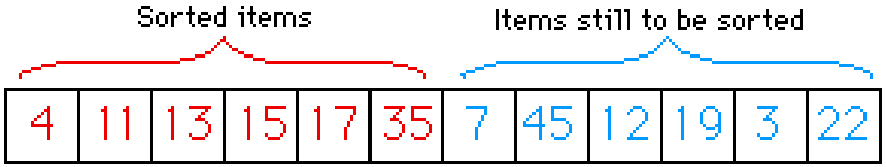


Temp: 15

Copy next unsorted item into Temp, leaving a "hole" in the array

Bump any items bigger than Temp up one space, then copy Temp into the "empty" location.

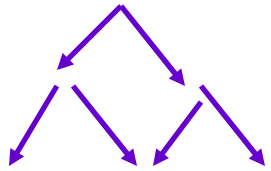




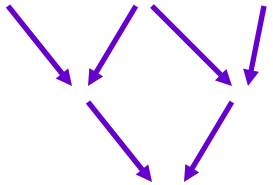
Now, the list of sorted items has increased in size by one item.

Mergesort: fast, N*log(N) in worst case

- **Idea**:
  - If *n<2* then the array is already sorted. Stop now.
  - Otherwise, *n>1*, and we perform the following three steps in sequence:
  1. Sort the left half of the the array.
  2. Sort the right half of the the array.
  3. Merge the now-sorted left and right halves.

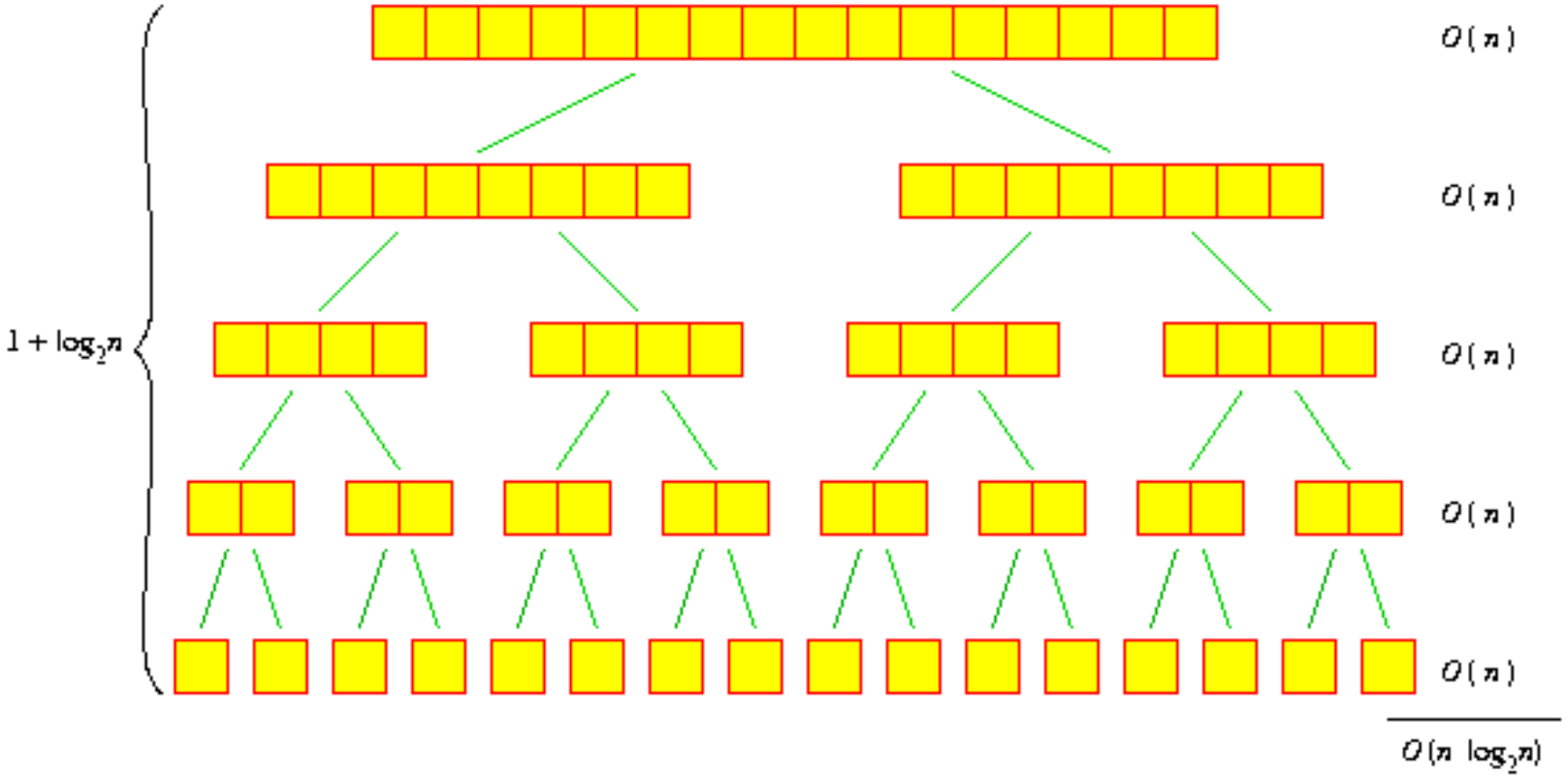


- **Ie:**
  - First phase, recursively split an array into small parts
  - Second phase: merge sorted parts, preserving right order!



- **Why it works fast:**
  - Merging two sorted arrays into one is simple and fast
  - Complexity (speed): X*X > 2 * ((X/2)*(X/2))

Explanation of how we get time N*log(N)



Mergesort: outer part of the algorithm

```
static void mergesort(int[] A) {
    if (A.length > 1) {
      // split the array into two pieces,
      // as close to the same size as possible.
      int[] first = extract(A, 0, A.length/2);
      int[] last  = extract(A, A.length/2, A.length);
      // sort each of the two halves recursively
      mergesort(first);
      mergesort(last);
      // merge the two sorted halves together
      merge(A, first, last);
    }
}
```

Mergesort: splitting function of the algorithm

```
// extract: returns a subarray of A,
// starting with index start in A,
// continuing to (but not including) index last.

static int[] extract(int[] A, int start, int last) {
   int[] ret = new int[last - start];
   for(int i = 0; i < ret.length; i++)
     ret[i] = A[start + i];
   return ret;
}
```

Mergesort: merging function of the algorithm

```
// merge: merges arrays a and b, placing the result into the
// array dest. This only works if both a and b are already in
// increasing order.
static void merge(int[] dest, int[] a, int[] b) {
  int i = 0;
  int j = 0;
  while(i < a.length && j < b.length) {
     if(a[i] < b[j]) {
        dest[i + j] = a[i];
        ++i;
     } else {
        dest[i + j] = b[j];
        ++j;
     }
  }
  for(; i < a.length; i++) dest[i + j] = a[i];
  for(; j < b.length; j++) dest[i + j] = b[j];
}
```

Quicksort: normally fast, N*log(N) in average case

- **Complexity:** N*log(N) in average case, but N*N in worst case. However, in practice typically a bit faster than always N*log(N) algorithms
- **Nice property:** does not need extra memory for temporary storage
- **Idea** is a bit similar to mergesort:
  - Split an array into two parts (not necessarily equal size) so that **all elements in first part are smaller than all elements of the second part**.
  - In order to do that, pick a probably-average-size element (pivot) to split an array into smaller-element and larger-element subarrays
  - Call quicksort recursively on both parts
  - Finally everything is sorted!!

- **NB!** Quicksort performs badly if we have no luck when choosing pivot. In worst case, one half of array contains just one element!

Quicksort: outer part of the algorithm

```
static void quicksort(int[] A, int lo, int hi) {
  if (hi <= lo) {
        // The list has length one or zero.  Nothing needs
        // to be done, so just return from the subroutine.
     return;
  }
  else {
     // Apply quicksortStep and get the pivot position.
     // Then apply quicksort to sort the items that
     // precede the pivot and the items that follow it.
     int pivotPosition = quicksortStep(A, lo, hi);
     quicksort(A, lo, pivotPosition - 1);
     quicksort(A, pivotPosition + 1, hi);
  }
}
```

Quicksort: inner part of the algorithm

```
static int quicksortStep(int[] A, int lo, int hi) {
    int pivot = A[lo];  // Get the pivot value.

    while (hi > lo) {
       while (hi > lo && A[hi] > pivot) hi--;
       if (hi == lo) break;
       A[lo] = A[hi];
       lo++;
       while (hi > lo && A[lo] < pivot) lo++;

       if (hi == lo) break;
       A[hi] = A[lo];
       hi--;
    } // end while
    A[lo] = pivot;
    return lo;
 }  // end QuicksortStep
```

Sorting: some comparisons

- We will have a look at the sorting applets running in parallel and Eck sorting lab timing view:

http://math.hws.edu/TMCM/java/xSortLab/

http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html